INDEPENDIO

CASE STUDY

# AI Project for Predictive Maintenance

Open-source data integration architecture for a nationwide electrical distribution network — replacing TIBCO EMS with Apache Kafka, with no cloud dependency and no per-connector licensing.

Independio · Enterprise Integration and AI Platforms on Open-Source Technology

01

## Executive Summary

The AI Project for Predictive Maintenance is a predictive maintenance and asset diagnostics platform built for a nationwide electrical distribution network. Its operation depends on continuously and reliably consolidating data from heterogeneous institutional systems, operational files, and field-deployed sensors, in order to feed the artificial intelligence models that prioritize inspection and maintenance across the network.

The platform's integration layer was built **entirely on Open Source components**, running inside the client's own infrastructure, with no dependency on cloud services and no licensing by connector, message volume, or CPU.

The centerpiece of this case is the replacement of the corporate messaging middleware — **TIBCO Enterprise Message Service (EMS)** — with **Apache Kafka** as the platform's integration bus. Message transformation, which in the commercial model is handled through proprietary graphical tools, was implemented with **custom-built Java components**, while extraction, transformation, and load processes run in **Python**.

> **Result:** an enterprise-grade integration architecture, with five distinct data-acquisition patterns running in production, with no licensing cost in the integration layer and no dependency on a single vendor.

02

## The Challenge

The platform had to solve five problems simultaneously — a combination rarely seen in a single project.

**Source heterogeneity.** The data required for asset diagnostics does not live in a single system. It comes from institutional SOAP and REST services, relational databases running on different engines, flat files generated by operational processes, IoT sensors across the network, and third-party monitoring systems that need to push information into the platform.

**Volume and continuity.** Field telemetry arrives continuously and cannot tolerate loss: a gap in the time series directly degrades the quality of the models' inference.

**On-premises operation.** Under client policy, all infrastructure runs inside the institutional network. There is no possibility of using managed cloud services for messaging, ingestion, or processing.

**Independence from the data model.** The platform had to be designed agnostic to the definition and structure of the source data, so that adding a new source would not require redesigning persistence or retraining the entire pipeline.

**Licensing constraints.** The corporate middleware available (TIBCO EMS) carried per-node and per-capacity licensing costs, along with a support model and release cycle disconnected from the project's own pace.

03

## Integration Architecture

The solution is organized into four decoupled layers. Each layer can evolve without affecting the others — this is the central property of the design.


## Layer 1 — Acquisition

One connector per source pattern, not one per system. This decision is what makes it possible to onboard a new system without writing new integration code: if the system exposes a REST API, it reuses the API connector; if it exposes a database, it reuses the SQL connector. Every connector publishes to the bus using a common envelope.

## Layer 2 — Integration bus

Apache Kafka acts as the single meeting point between producers and consumers. No connector knows its final consumer, and no transformation process knows its origin. Topics are organized by functional domain rather than by source system, so replacing a source system doesn't require modifying consumers.

Kafka's configured retention enables two capabilities that the previous scheme required additional development to achieve: **historical reprocessing** — re-running a corrected transformation over data already received — and **simultaneous onboarding of new consumers** — adding a new analytical model that consumes available history without touching the original source.

## Layer 3 — Processing

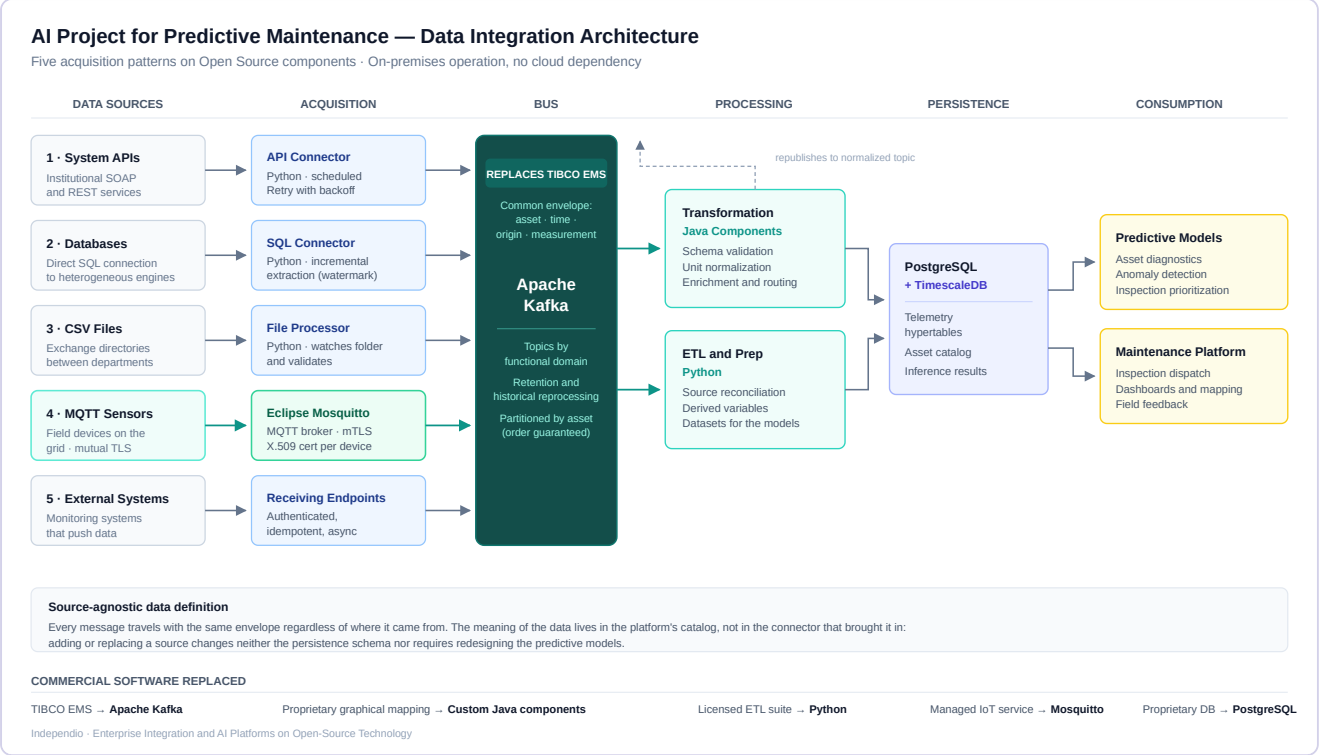
Two technologies with separate responsibilities: **custom-built Java components** for message transformation — schema validation, normalization of units and formats, enrichment with asset catalogs, and routing between topics — replacing the function that adapters and graphical mapping cover in the commercial middleware ecosystem; and **Python** for ETL processes — scheduled extraction, reconciliation across sources, calculation of derived variables, and preparation of the datasets consumed by the predictive models. This choice reflects that the same language sustains the models' full lifecycle, eliminating the boundary between data engineering and data science.

## Layer 4 — Persistence and consumption

PostgreSQL as the primary database, with TimescaleDB handling telemetry time series via hypertables. Telemetry, asset catalogs, and inference results coexist in a single engine, making it possible to resolve in SQL queries that would otherwise require federating across separate systems.

**Figure 1**

# Data Integration Architecture — Full Diagram




*Five acquisition patterns feeding a common Kafka bus, processed and persisted for consumption by predictive models and the maintenance platform.*

04

# Ingestion Patterns Implemented

## 4.1 Consuming institutional system APIs

**MECHANISM.** Scheduled Python clients that query the client's SOAP and REST services on a cadence defined per domain.

**HANDLING.** Each response is converted to the common envelope and published to the corresponding topic. Service errors do not halt the pipeline: the connector retries with exponential backoff and logs the failure as an event, so that a source's temporary unavailability is observable rather than silent.

**WHY IT MATTERS.** This is the most common pattern in enterprise integration, and the one commercial suites license by connector. Here, there is no connector to license.

## 4.2 Direct database connections

**MECHANISM.** Incremental extraction across heterogeneous relational engines using standard drivers. Per-table watermarks are used to read only the delta since the last run.

**HANDLING.** Extraction never writes to the source database and runs in windows agreed with each system's owners, with concurrency limits so as not to compete with transactional operations.

**WHY IT MATTERS.** It allows onboarding systems that don't expose services, without requiring their owners to build an API.

### 4.3 CSV file processing

**MECHANISM.** Monitoring of exchange directories. When a new file is detected, its structure is validated, it's processed in batches, and archived along with its result.

**HANDLING.** Each file is treated as a transaction: if a row fails validation, it's isolated in a rejection log with the reason, and the rest of the batch continues. The original file is preserved, allowing reprocessing without having to request it again from the originating team.

**WHY IT MATTERS.** The flat file remains the real mechanism of exchange between departments in large organizations. Ignoring it forces those departments to change how they work; supporting it well removes that friction.

### 4.4 MQTT telemetry from field sensors

**MECHANISM.** An **Eclipse Mosquitto** broker deployed inside the client's infrastructure, with mutual TLS and an X.509 certificate per device. Sensors publish into a per-device topic hierarchy (`project/{device_id}/...`) with access-control rules that prevent one device from publishing into another's namespace.

**HANDLING.** A bridge consumes from the broker and publishes to Kafka, at which point telemetry is persisted and available to multiple consumers. The separation between the MQTT broker and the event bus is deliberate: Mosquitto handles the field protocol and device authentication; Kafka handles durability and internal distribution.

**WHY IT MATTERS.** This replaces a managed IoT ingestion service with a broker the client owns, running inside the institutional network with no transit over the public internet.

### 4.5 Receiving endpoints for monitoring systems

**MECHANISM.** Endpoints exposed by the platform itself so that other network monitoring systems can push information, with authentication and idempotency keyed by event identifier.

**HANDLING.** The endpoint validates the request, acknowledges receipt, and publishes to the bus. The database write happens afterward, asynchronously, so that a spike in submissions doesn't translate into rejections toward the sending system.

**WHY IT MATTERS.** This reverses the direction of integration. The platform stops being purely a consumer and becomes a receiving endpoint too, without coupling its availability to that of the sender.

05

## Source- and Model-Agnostic Data Definition

The hardest property of this architecture to replicate isn't technical — it's a design choice: **the definition of the data is decoupled from its origin.**

Every message entering the bus — whether it originated from a SOAP service, a SQL query, a CSV row, or a sensor — travels with the same envelope: asset identifier, timestamp, provenance, measurement type, and payload. The meaning of the data lives in the platform's catalog, not in the connector that brought it in.

This produces three concrete operational consequences:

**Onboarding sources without redesign.** Adding a new system means configuring an existing connector and mapping its fields to the catalog. Neither the persistence schema nor the consumers are modified.

**Replacing sources without impact.** If the client replaces one institutional system with another, the change is absorbed entirely in the acquisition layer. The predictive models never notice.

**Independent evolution of the models.** The AI models consume from the catalog, not from the sources. A change in analytical technique doesn't require touching the integration layer, and a change of source doesn't require retraining for formatting reasons.

06

# Replacing Commercial Software

| LAYER | COMMERCIAL SOLUTION | OPEN-SOURCE ALTERNATIVE | BENEFIT ACHIEVED |
|---|---|---|---|
| Messaging middleware | **TIBCO EMS** | **Apache Kafka** | No licensing by node or by added capacity. Adds event retention and reprocessing, a capability the queue-based model didn't offer. |
| Message transformation | Adapters and graphical mapping in the proprietary middleware | **Custom-built Java components** | Transformation logic versioned in the project's repository, with automated tests. No dependency on a graphical tool or certified specialists in it. |
| ETL / data integration | Commercial ETL suite licensed by connector | **Python** | No cost per connected source. Language continuity with the analytics/modeling layer. |
| IoT ingestion | Managed cloud IoT service | **Eclipse Mosquitto** | Runs inside the institutional network. No cost per message or per connected device. |
| Database | Proprietary relational engine | **PostgreSQL + TimescaleDB** | No per-core licensing. Time-series and relational data in a single engine. |

## Considerations in migrating from TIBCO EMS to Kafka

This substitution wasn't a piece-by-piece replacement, because the two messaging models differ fundamentally. The points that shaped the design:

- **From queue to log.** EMS delivers and discards; Kafka retains. Consumers stopped being the exclusive recipients of a message and became independent readers of a log, each tracking its own offset. This made it possible to add analytical consumers without negotiating with existing ones.
- **Delivery and ordering.** Order is guaranteed per partition, which required defining the partition key by asset in order to preserve the event sequence for a given network element.
- **Consumer-side idempotency.** Under at-least-once delivery semantics, deduplication was resolved on the consumer side using the event identifier, not in the bus itself.
- **Observability.** Per-group consumer lag became the primary operational health indicator for the integration layer, replacing the queue-depth monitoring used in the previous scheme.

07

# Results

**Note:** the indicators below are pending confirmation with verified figures from the project before this document is published or shared externally. No values have been estimated or assumed.

| INDICATOR | VALUE |
|---|---|

| | |
|---|---|
| Sources integrated in production | *Pending confirmation* |
| Field devices publishing telemetry | *Pending confirmation* |
| Daily volume of messages processed | *Pending confirmation* |
| Annual savings in integration-layer licensing | *Pending confirmation* |
| Time to onboard a new source | *Pending confirmation* |
| Platform availability | *Pending confirmation* |

### Qualitative results, verifiable without a figure

- The platform runs in production on the client's own infrastructure, with no dependency on cloud services.
- The integration layer generates no licensing cost per connected source, per device, or per message volume.
- Onboarding new data sources requires no changes to the persistence model or the analytical models.
- All integration and transformation logic is owned by the project, version-controlled, and auditable, with no black-box components.

08

## Lessons Applicable to Other Migrations

**Middleware isn't migrated — it's redesigned.** Moving a queue topology onto an event bus without revisiting the consumption model just reproduces the original system's limitations and wastes what the replacement offers. A successful migration starts from an inventory of flows, not an inventory of queues.

**One connector per pattern, not per system.** This is the decision that determines the cost of growth. Commercial suites charge per connector precisely because the alternative model — one generic connector per pattern — drastically reduces the number of moving parts.

**Decoupling the meaning of the data is what sustains the long term.** It's the difference between an integration layer that ages along with its sources and one that outlives their replacement.

**Language continuity between data and models.** When ETL and the models share a language and ecosystem, the reimplementation gap between the analytical prototype and the production process disappears.

**The replacement should be measured in capability, not just cost.** The strongest argument with the client wasn't the licensing savings — it was that the resulting architecture does things the previous one couldn't: historical reprocessing, adding consumers without coordination, and full traceability of data lineage.

09

## Independio Services

This case corresponds to the scope of services Independio offers for replacing commercial integration platforms.

### Assessment and diagnostics

Inventory of integration flows, licensing dependencies, and layer-by-layer replacement feasibility analysis, with effort and risk estimated per flow.

### Target architecture design

Definition of the event bus, topic model, partitioning strategy, delivery guarantees, and a source-agnostic data catalog.

### Implementation and migration

Development of pattern-based connectors, transformation components, ETL processes, and parallel operation through to final cutover.

### Knowledge transfer and support

Training for the client's team, operational documentation, and support through stabilization.

---

**INDEPENDIO**

Enterprise Integration and AI Platforms on Open-Source Technology